Reinforcement Learning
Exercise 1

September 6, 2022

Introduction

In this exercise we will take a first look at a reinforcement learning environment, its components and modify the reward function of a simple agent.

States, observations and reward functions

The provided Python script (`train.py`) instantiates a *Cartpole* environment from OpenAI gym and an RL agent that acts in it. The `agent.py` file contains the implementation of a simple reinforcement learning agent; for the sake of this exercise, you can assume it to be a black box (you don't need to understand how it works, although you are encouraged to study it in more detail). You don't have to edit the `agent.py` file to complete this exercise session. Another environment, called *Reacher* is implemented in file `reacher.py`, and (incomplete) code to help visualise reward functions is found in file `plot_rew.ipynb`. The folder `cfg` contains config files to e.g. define the maximum number of steps in an episode. **In this exercise you will need to edit the Python file `reacher.py`, and the Jupyter Notebook file `plot_rew.ipynb`.**

Cartpole

The *Cartpole* environment consists of a cart and a pole mounted on top of it, as shown in Figure 1. The cart can move either to the left or to the right. The goal is to balance the pole in a vertical position in order to prevent it from falling down. The cart should also stay within limited distance from the center (trying to move outside screen boundaries is considered a failure).

The state and the observation are four element vectors:

Aalto University
School of Electrical Engineering

Reinforcement Learning course staff
Aalto Robot Learning Lab
aalto.fi, rl.aalto.fi

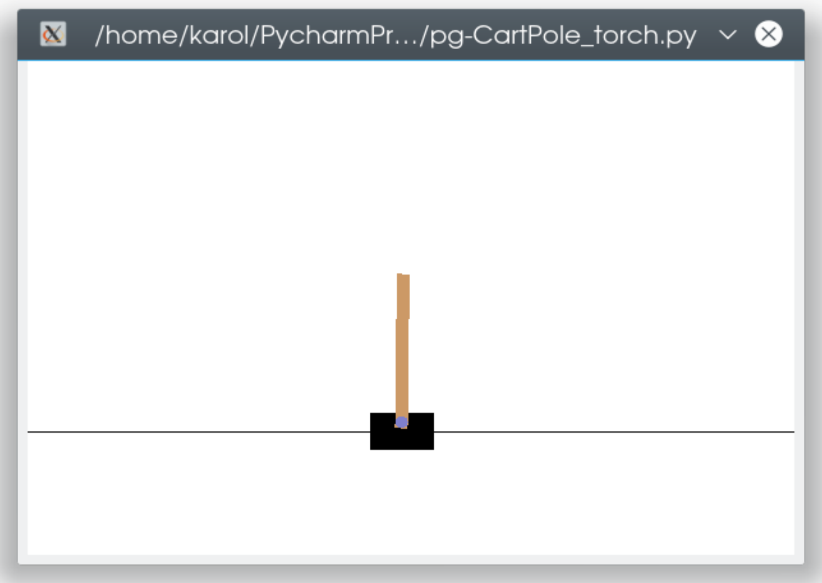

Figure 1: The Cartpole environment

$$o = s = \begin{pmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{pmatrix}, \tag{1}$$

where x is the position of the cart, $\dot{x}$ is its velocity, θ is the angle of the pole w.r.t. the vertical axis, and $\dot{\theta}$ is the angular velocity of the pole.

In the standard formulation, a reward of 1 is given for every timestep the pole remains balanced. Upon failing (the pole falls) or completing the task, an episode is finished.

The `train.py` script will record videos of the agent's learning progress during training, and the recorded videos are saved to `results/video/CartPole-v0/train`. By default, the training information is saved to `results/logging/CartPole-v0_logging.pkl` as well as on cloud via wandb. When the training is finished, the models are saved to `results/model/Cartpole-v0_params.pt`. The models can be tested by setting `testing=true`, and if the models are saved to a different path, you can use `model_path=<YOUR MODEL PATH>` to indicate it. Videos of the agent's behaviour during testing are saved to `results/video/CartPole-v0/test`.

Task 1 - 10 points Train a model with **100** timesteps per episode. Use the command `python train.py seed=1 max_episode_steps=100`. Then test the model for **1000** timesteps with command: `python train.py testing=true seed=1 max_episode_steps=1000 use_wandb=false` – this will evaluate the trained model in 10 episodes and report the average reward (and episode length) for these 10 episodes. **Export the training plot episodes-ep_reward from wandb (see README.md) and attach the plot in your report. Report also the average reward after testing the model.**

Question 1.1 — 10 points Test trained model from Task 1 **five** times with different random seeds. Did the same model, trained to balance for 100 timesteps, learn to always balance the pole for 1000 timesteps? Why/why not?

Task 2 - 10 points Repeat the experiment in Task 1 five times, each time training the model from scratch with 100 timesteps and testing it for 1000 timesteps (in this Task it is enough to test with one seed). Use a different seed number for each training/testing cycle. **Report the average test reward for each repeat.**

Question 2.1 — 15 points: Are the behavior and performance of the trained models the same every time? Why/why not? Analyze the causes briefly.

Question 2.2 — 10 points: What are the implications of this stochasticity, when it comes to comparing reinforcement learning algorithms to each other? Please explain.

Reacher

Now we will focus on designing a reward function for a different environment, the *Reacher* environment, where a two-joint manipulator needs to reach a goal (see Figure 2).

The Cartesian (x, y) position of the end-effector of the manipulator can be determined following the equation:

$$\begin{aligned} x &= L_1 \sin(\theta_0) + L_2 \sin(\theta_0 + \theta_1), \\ y &= -L_1 \cos(\theta_0) - L_2 \cos(\theta_0 + \theta_1), \end{aligned} \tag{2}$$

where $L_1 = 1$, $L_2 = 1$ are the lengths, and θ_0, θ_1 the joint angles of the first and second links respectively.

The state (and observation) in this environment is the two element vector

$$o = s = \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix}, \tag{3}$$

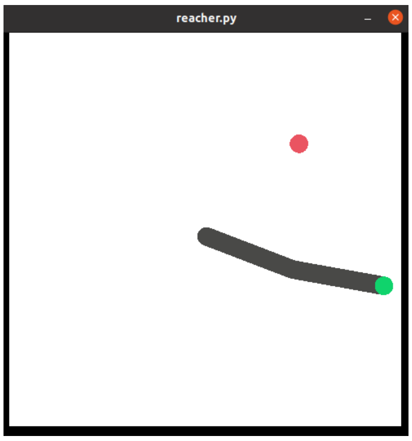

Figure 2: The Reacher environment

The action space now consists of 5 "options"; 4 correspond rotating the first/second joint left/right, and the final one performs no motion at all (the configuration doesn't change). The episode terminates when the agent reaches the target position, marked in red. To run the training script with the Reacher environment, use `python train.py env=reacher_v1`.

Now, let us design a custom reward function and use it for training the RL agent.

Task 3 — 20 points: Edit the function `get_reward` in `reacher.py`, and write a reward function to incentivise the agent to learn the following behaviors:

1. Keep the manipulator rotating clockwise continuously (wrt angle θ_0). You can use a lower number of training episodes for this, e.g. `train.py env=reacher_v1 train_episodes=200`
2. Reach the goal point located in $\mathbf{x} = [1.0, 1.0]$ (marked in red). Use at least 500 training episodes.

Train one model for each behavior and include the model files in your submission. Also include the reward function implementations in your report (write them out or attach as screenshots).

Hint: Use the observation vector to get the quantities required to compute the new reward (such as the position of the manipulator). You can get the Cartesian position of the end-effector with `env.get_cartesian_pos(state)`.

Task 4 — 10 points: Now, let us visualize the reward function for the second behavior (reaching the goal [1,1]). Plot the values of the second reward function from Task 3 and the learned best action as a function of the state (the joint positions). Use the `plot_rew.ipynb` script as a starting point. After plotting, answer the questions below. **Include the two figures produced by plot_rew.ipynb in your report.**

Question 4.1 — 5 points: Where are the highest and lowest reward achieved?

Question 4.2 — 10 points: Did the policy learn to reach the goal from every possible state (manipulator configuration) in an optimal way (i.e. with lowest possible number of steps)? Why/why not?

Feedback

In order to help the staff of the course as well as the forthcoming students, it would be great if you could answer to the following questions in your submission:

- How much time did you spend solving this exercise?
- Did you find any of the particular tasks or questions difficult to solve?

Submitting

The deadline to submit the solutions through MyCourses is on Sep. 19, 23:55. Example solutions will be presented during exercise sessions the following Monday.

Your submission should consist of (1) a **PDF report** containing **answers to the Questions** asked in these instructions and plots/model files/reported metrics **as required in each of the Tasks**, (2) **the code** with solutions used for the exercise. Please remember that submitting a PDF report without following the **Latex template** provided by us will lead to subtraction of points.

For more formatting guidelines and general tips please refer to the README.md.

If you need help or clarification solving the exercises, you are welcome to join the Slack channel and exercise sessions.